11-5-1992

Compiling ACE for Distributed-Memory Machines

Jun Song
Portland State University

AN ABSTRACT OF THE THESIS OF Jun Song for the Master of Science in Computer Science presented November 5, 1992.

Title: Compiling ACE for Distributed-Memory Machines

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

Jingke Li, Chair

Leonard D. Shapiro

Michael A. Driscoll

Distributed-memory machines offer a very high level of performance, flexibility and scalability. But the memory organization of this kind of machine determines that processes on different processors must communicate explicitly by sending and receiving messages. As a result, the programmer faces the enormously difficult task of detailed planning of algorithm-irrelevant, low-level communication issues. This level of programming resembles writing assembly programs for a sequential machine.

ACE is a message-passing language with abstract communication statements. It was defined by Dr. Jingke Li at Portland State University. The communication in ACE is still explicit, but it is abstracted to a higher level. The abstraction can help balance the needs of ease of programming and high performance.

This thesis discusses how those high-level communication abstractions can be transformed into low-level communication routines. It presents the design and implementation of a compiler that transforms an ACE program into a C program with low-level communication routines. The compiler is implemented for the Intel iPSC/2 hypercube multiprocessor machine. Compared to their low-level counterparts, ACE programs are easier to write and are more understandable. Compared to their high-level counterparts, more efficient code can be generated since the communication information is expressed explicitly in ACE and the compiler itself is much less complex. ACE also enables the users to fine tune some critical communication segments. Some well known parallel algorithms written in ACE are compiled by the compiler as examples, and experimental results of their performance are included.

COMPILING ACE

FOR DISTRIBUTED-MEMORY MACHINES

by

JUN SONG

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

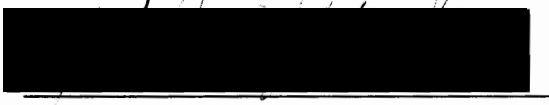
Portland State University
1992

TO THE OFFICE OF GRADUATE STUDIES:

The members of the committee approve the thesis of Jun Song presented November 5, 1992.

Jingke Li, Chair

Leonard D. Shapiro



Michael A. Driscoll

APPROVED:

Leonard D. Shapiro, Chairman, Department of Computer Science

Roy W. Koch, Vice Provost for Graduate Studies and Research

TABLE OF CONTENTS

PAGE

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

I INTRODUCTION 1

I.1 ACE 3

I.2 TARGET MACHINE 4

I.2.1 Machine Architecture 4
I.2.2 Programming Environment 5
I.2.3 Message Passing 5

I.3 RELATED WORK 6

I.4 ORGANIZATION OF THE THESIS 8

II ACE 9

II.1 VIRTUAL PROCESSOR DOMAIN 9

II.2 DATA DISTRIBUTION 10

II.2.1 Alignment 10
II.2.2 Distribution 11

II.3 DATA MOVEMENT 12

II.4 PARALLEL CONSTRUCT 16

II.5 COLLECTIVE COMMUNICATION LIBRARY 16

III COMPILATION APPROACH 19

III.1 SYNTAX TREE AND SYMBOL TABLE 21

III.2 DECLARATION STATEMENT TRANSFORMATION . . . 23

III.3 COMMUNICATION STATEMENT TRANSFORMATION . 27

III.3.1 Local Variable Initialization 27
III.3.2 Communication Patterns 30
III.3.3 Code Structure 31
III.3.4 Data Packing and Unpacking 34
III.3.5 Optional Domain Predicate 38

III.4 PARALLEL LOOP TRANSFORMATION 40

III.5 INPUT/OUTPUT 41

III.6 TARGET CODE 44

III.7 SEQUENTIAL TRANSFORMATION 46

III.7.1 Data Distribution 47
III.7.2 Data Movement 47
III.7.3 Parallel Loops 49

IV EXPERIMENT 51

IV.1 CANNON'S ALGORITHM 52

IV.2 GAUSSIAN ELIMINATION 53

IV.3 PERFORMANCE ANALYSIS 56

IV.3.1 Performance Measurement 56
IV.3.2 Analysis 58

V CONCLUSION 62

V.1 RESEARCH SUMMARY 62

V.2 FUTURE WORK 63

REFERENCES 64

APPENDICES

A TARGET CODE FOR CANNON'S ALGORITHM 65

B TARGET CODE FOR GAUSSIAN ELIMINATION 70

LIST OF TABLES

TABLE PAGE

I Performance of Cannon's Algorithm 59

II Performance of Gaussian Elimination 61

LIST OF FIGURES

FIGURE		PAGE
1	Block and Cyclic Distribution.	12
2	Parallelize Matrix Multiply Using `forall` Loop.	17
3	The Evolution of An ACE Program To Target Code.	20
4	Symbol Table Entry of A Domain Variable.	22
5	Symbol Table Entry of A Data Variable.	23
6	Example of Declaration.	26
7	Local Variable Initialization.	29
8	Four Basic Communication Patterns.	31
9	One-to-One Communication.	32
10	One-to-Many Communication.	34
11	Many-to-One Communication.	35
12	Many-to-Many Communication.	35
13	Problem With Packing And Unpacking Row Major.	38
14	Usage of `input` And `output`.	44
15	Target Code of `input` And `output`.	45
16	Sequential Target Code of Different Data Representation.	49
17	Illustration of Cannon's Algorithm.	54
18	Cannon's Algorithm In ACE.	55
19	Gaussian Elimination In ACE.	57
20	Speedup of Cannon's Algorithm On iPSC/2.	60
21	Speedup of Gaussian Elimination On iPSC/2.	60

CHAPTER I

INTRODUCTION

Distributed-memory multicomputers offer a very high level of performance, flexibility and scalability. Potentially, a massively parallel machine can out-perform a sequential machine of the same cost by several orders of magnitude. But faster hardware does not guarantee that faster, more efficient programs will follow. The memory organization of distributed-memory machines determines that processes must communicate explicitly by sending and receiving messages. As a result, the programmer faces the enormously difficult task of detailed planning of computation. The successful approach so far has been to program them directly in a message passing system. This approach requires the user to control explicitly algorithm-irrelevant, low-level issues in application programs. This level of programming resembles writing assembly programs for a sequential machine.

A distributed-memory machine has a set of identical processors linked by an interconnecting network. Each processor is tightly coupled to a memory unit that is physically separate and logically private from the memory units of all other processor. A global shared-memory does not exist. The fixed numbering of the nodes, $0, 1, \ldots, N-1$, together with the unique numbering of the processes within each processor, establishes globally unique identifiers for processors; hence, a global name space.

Data sharing among processors is achieved through explicit communication. Interprocess communication occurs by routing messages through networks such as

binary n-cube or mesh. The networks are also extensible to allow for systems with different numbers of processors.

Writing efficient programs for distributed-memory machines is a great challenge for a programmer. Many issues that do not arise in programming shared-memory machines must be addressed in programming distributed-memory machines:

- Data Layout

  Since there is no global shared memory on a distributed-memory machine, large data structures in an application must be partitioned and distributed over the processors.

- Communication

  In order to share a piece of data, explicit message passing must be constructed and inserted into the user program.

- Parallelism

  Parallelism in the application must be made explicit in the user program. Generally speaking, this involves breaking a computation into a collection of *parallel tasks*, which are assigned to different processors and executed in parallel.

Many research projects focus on building smart compilers that will enable users to program a distributed-memory machine just like they would do on a shared-memory machine [4, 5, 9, 14, 16]. In this thesis, we are trying to reach a less ambitious goal. Dr. Jingke Li has defined a message-passing language for distributed-memory machines, ACE [11]. The communication in ACE is still explicit, but it is abstracted to a higher level. The abstraction can help balance the needs of ease of programming and high performance. The communication is described as data move-

ment on a virtual processor domain while data is addressed in global space. ACE is designed in a way that it can also be used as an intermediate language. It can act as a middle step of compilations of higher level parallel languages with implicit communication, such as Fortran 90. This thesis will discuss how those high-level abstractions of communication can be transformed into low-level communication routines, and how an ACE program can be compiled to a low-level message passing program that can be executed on a target machine.

I.1 ACE

ACE is language that is able to address the above points in a program. It is based on C with abstract communication extensions, hence the name, ACE.

All data arrays that are to be distributed are aligned to virtual processor domains. An ACE program will distribute the data to all node processors according to how they are aligned to the virtual processor domain. Communication between processors can be expressed using a set of high level abstract routines. These abstract communication routines greatly ease the writing of message-passing programs. Furthermore, they can be used to illustrate some efficient parallel algorithms that are difficult for conventional languages to address. ACE programs are easier to write and more understandable than low-level message-passing ones.

For each abstract communication routine, a corresponding routine is written using the actual message-passing scheme of the target machine. These routines make up our Collective Communication Library (CCL). The implementation of the CCL on different target machines will make ACE portable across these machines.

Comparing to its low-level counterpart, ACE programs are easier to write and understand. The task of initiating and coordinating of detailed message-passing which is essential to low-level programs is greatly reduced in ACE. The algorithm

that might be buried in so many communication-related statements in a low-level program will be more explicit in an ACE program. The abstraction of ACE makes debuggin easier, too.

ACE can also be viewed as an intermediate language for compiling other higher level (implicit data distribution and communication) languages.

I.2 TARGET MACHINE

The target machine under consideration for this research project is Intel's iPSC/2 [7].

I.2.1 Machine Architecture

The iPSC/2 system is an ensemble of processing nodes, each connected to the other via message passing on the network. Solutions to problems are computed by solving a portion of the problem on each of the processing nodes. The system consists of four main functional components:

- The cube

  The cube provides the multiprocessor computational power of the iPSC/2 system. Software commands let users "partition" this cube into one or more subsets of nodes.

- The node

  Each iPSC/2 node contains a 32-bit microcomputer with the execution speed and memory capacity of a typical superminicomputer. Each node is also equipped with a Direct-Connect routing module for high-speed message passing within the system's hypercube communication network.

- The system resource manager

The system resource manager (SRM) serves as the iPSC/2's connection to the outside world. It has three main purposes: administrative console for the system, gateway to other computers and workstations connected by Ethernet to the system and host for various development tools.

- The network

  Although physically implemented as a hypercube, the Direct-Connect communications hardware and operating system software allow users to assume communications as if every node of the system were directly connected to each other.

I.2.2 Programming Environment

The iPSC/2 system supports FORTRAN, C, and Common LISP languages.

The C language is a general purpose language which features economy of expression, modern control flow and data structures, and a variety of operators. It is based on the standard Kernighan and Ritchie definition of the C programming language. The C compiler is the Green Hills C compiler.

The standard C language is augmented by a library of pre-defined functions that allow the C programmer to implement the message passing required to build concurrent applications for the iPSC/2. These functions allow C processes to send and receive messages, probe for messages of a given type, and to obtain ancillary information about messages and the iPSC/2 system.

I.2.3 Message Passing

The iPSC/2 is a distributed-memory machine. Nodes communicate with each other and with the host via message passing. A message may be up to 256 Kbytes long if data is being passed to or from the host. It may be of unlimited length if it

is being passed from node to node. Messages may be of any format or data type. A sending process may send a message of a particular data type. It is the user's responsibility to ensure the receiving process expects a message of that type.

A message can be sent from one process and received by one or more processes on the same node, or on different nodes. From a programming point of view, sending a message sends a copy of the contents of a buffer from one process to another. The buffer variable may be of any data type including the C data type "`struct`", allowing any amount of data to be sent in one variable.

I.3 RELATED WORK

To program a distributed-memory machine so that the machine architecture can be well matched and utilized is a difficult task. An obvious approach is to program each processor of the machine in a sequential language and insert communication statements in the program for sending and receiving messages. Almost all of the commercial machines have languages to support this programming model. For instance, Intel's hypercubes supports C and Fortran with message passing extensions. However, programming in this fashion is tedious and error-prone. The programs written this way are usually machine dependent and are not portable across different machines.

Over the years, researchers have proposed many solutions to this problem.

- Fortran D

  Fortran D [2, 3] is a version of Fortran enhanced with data decomposition specifications. The `DECOMPOSITION` statement is used to declare a problem domain for each computation; the `ALIGN` statement is used to specify how arrays should be aligned with respect to one another; and the `DISTRIBUTE`

statement is used to map the problem domain to the physical machine. A Fortran D compiler for the iPSC/860 is under construction [4, 5].

- Fortran 90

  Fortran 90 extends Fortran 77 with a set of parallel constructs and intrinsic functions. The parallel constructs support whole array operations and array sections, which simplifies the writing of data parallel applications. A proposal for compiling Fortran 90 programs for distributed-memory machines can be found in [16].

- Dataparallel C

  Dataparallel C [6] is a SIMD extension to the standard C programming language. It is derived from the original C* language developed by Thinking Machine Corporation [15]. The user specifies parallel computations as *actions* on a *domain*. The compiler automatically determines the data distribution and generates communications.

- Crystal

  Crystal [1] is a functional language designed to provide a convenient means for expressing parallelism and locality. It contains special constructs for representing data parallel computations. A Crystal compiler is developed for distributed-memory machines [9]. A Crystal program is first transformed to a shared-memory parallel program and then to a message-passing program.

- MetaMP

  MetaMP language [13] consists of normal, sequential C and MetaMP directives which modify the meaning of the sequential `for` loops to their parallel, distributed-memory counterparts.

Most of the languages mentioned above give their compilers some guidance of what data arrays are to be distributed and how to distribute them. The major forms of parallelism are parallel loops over arrays. They are at a relatively high level as far as communication is concerned. There is no explicit communication in those languages. The communication is generated by the compiler with respect to the reference patterns in the source program.

I.4 ORGANIZATION OF THE THESIS

The rest of this thesis is organized as the following: Chapter II introduces the language ACE, the abstract communication statements and the collective communication library. Chapter III presents the design and the implementation of the ACE compiler. Three programming examples are given in Chapter IV, a matrix multiply program, a matrix multiply program using Cannon's algorithm, and a Gaussian elimination program. Chapter V is a summary of the work has been done and a brief discussion of future work.

CHAPTER II

ACE

ACE is a message-passing language with high-level abstract communications [11]. It is based on C with extensions in the following three areas: data distribution, data movement and parallel loops. We do not intend to describe the language in its full extent here. We will only introduce major ACE features that are relevant to this project. Interested readers are encouraged to read [11].

Section II.1 discusses the concept of virtual processor domain. How to express data distribution is discussed in Section II.2. High-level abstract communication statements and their low-level equivalent are presented in Sections II.3 and II.5. Section II.4 describes how parallelism is expressed in ACE.

II.1 VIRTUAL PROCESSOR DOMAIN

For describing both computations and communications before data distribution, we define an abstract machine with a global addressing space, a *virtual processor domain* (or simply *domain*). A virtual processor domain models a parallel machine with processors arranged in a multidimensional grid. A one dimensional domain is specified by

domain $D = [1 : n]$

where n is the number of virtual processors (VPs) in the domain. A two-dimensional domain is specified by

domain $D = [1 : n_1, 1 : n_2]$.

Both parallelism and communications are expressed explicitly with respect

to domains. Parallelism is made explicit through parallel constructs such as `forall`. Communications are explicit in forms of data movement over virtual processor domains.

II.2 DATA DISTRIBUTION

Data distribution information is essential to a message-passing program. Without information about how data is distributed across the processor network, it is impossible to specify communication explicitly. In ACE, data distribution is done in two steps. First, data arrays are aligned to a virtual processor domain using the `align` statement. Second, this virtual processor domain is mapped onto a physical processor network using the `distribute` statement.

II.2.1 Alignment

Alignment is the process of aligning a group of data arrays onto a single virtual processor domain. In ACE, the following four alignments can be specified: *permutation, embedding, replication* and *collapse.* Each alignment is represented by an alignment declaration statement.

- Permutation

  It aligns an array to a domain of the same rank. The corresponding dimensions must have the same size.

  align $a[i]$ to $[i] : D_1$;
  align $b[i][j]$ to $[i, j] : D_2$;
  align $b[j][i]$ to $[i, j] : D_2$;

- Embedding

  It aligns an array to a domain of high rank. The matching dimensions must have the same size and the other dimensions of the domain must be aligned

with constants.

align $a[i]$ to $[i, 0]$: D_2
align $a[i]$ to $[2, i]$: D_2

- Replication

  It replicates an array along a dimension of a domain. The replication dimension is denoted by a colon (":"). A scalar can also be replicated.

  align $a[i]$ to $[i, :]$: D_2
  align s to $[:]$: D_1

- Collapse

  It aligns an array to a domain of lower rank by collapsing its dimensions. The collapsed dimensions are denoted by colons.

  align $a[:]$ to $[:]$: D_1
  align $b[i][:]$ to $[i]$: D_1

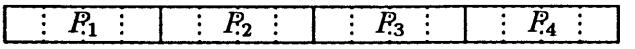
  The first statement declares that array a is replicated on all nodes in domain D_1.

II.2.2 Distribution

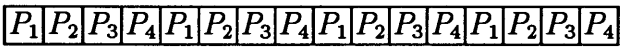
In ACE, we use the `distribute` statement to specify the mapping of the virtual processor domain to the physical processor network. There are two types of distributions in ACE, block and cyclic. Suppose there are P physical processors and N elements in a virtual processor domain D. We assume for simplicity that P divides N evenly. The distributions can be described as follows:

- Block

  distribute D over $[P]$;



(a) Block distribution



(b) Cyclic distribution

Figure 1. Block and Cyclic Distribution.

Divide the domain into contiguous chunks of size N/P, assigning one block to each processor (Figure 1(a)).

- Cyclic

  distribute D **over** $[P]$ cyclic;

  A round-robin division of the domain, assigning every P^{th} element to the same processor (Figure 1(b)). Cyclic distributions are useful for load balancing.

II.3 DATA MOVEMENT

High level data movement in a virtual processor domain is described by abstract communication statements. We selected a set of basic and common communication routines. They can be used in programs to describe communications over virtual processor domains. The communication statements are defined with respect to a single domain. We use the following notation

with $(idx : domain)$ $key(\alpha@source \rightarrow \beta@dest)$;

to describe an abstract communication statement on a virtual processor domain. The clause $(idx : domain)$ specifies the domain; the expression $\alpha@source$ specifies the source data and its virtual processor; and the expression $\beta@dest$ specifies the destination buffer and its virtual processor.

The set of communication statements currently exists in ACE are explained as follows:

- Copy

  with $([i] : D)$ **copy** $(a[c_1]@[c_1] \rightarrow b[c_2]@[c_2])$;

  It copies the value of $a[c_1]$ to $b[c_2]$.

- Swap

  with $([i] : D)$ **swap** $(a[c_1]@[c_1] \rightarrow b[c_2]@[c_2])$;

  It exchanges the values of $a[c_1]$ and $b[c_2]$.

- Spread

  with $([i] : D)$ **spread** $(a[c_1]@[c_1] \rightarrow b[i]@[i])$;

  The value of $a[c_1]$ is broadcast to all VPs in domain D and stored in $b[i]$.

- Scatter

  with $([i] : D)$ **scatter** $(a[:]@[c_1] \rightarrow b[i]@[i])$;

  The elements of data array a on VP c_1 is scattered around all VPs in domain D and stored in $b[i]$.

- Reduce

  with $([i] : D)$ **reduce** $(a[i]@[i] \rightarrow b[c_2]@[c_2], op)$;

  It reduces all elements of array a on all VPs with binary operation op. The result is stored in $b[c_2]$. op specifies a binary operation such as '+' or '*'.

- Gather

  with $([i] : D)$ **gather** $(a[i]@[i] \rightarrow b[:]@[c_2])$;

  This has the opposite effect of **scatter**. Elements of a on all VPs are collected to VP c_2, assembled and stored in b. It is roughly equivalent to a **reduce** with a concatenation operator.

- Search

  with $([i] : D)$ **search** $(a[i]@[i] \rightarrow b[c_2]@[c_2], loc, op)$;

  This statement conducts a binary search among elements of a on all VPs with a binary function *op*. It finds out the value and the location of the winner and stores them in $b[c_2]$ and *loc*. *op* is a flag of one of the following values, *MIN* (minimum), *MAX* (maximum), *MINABS* (minimum of the absolute value) and *MAXABS* (maximum of the absolute value).

- Shift

  with $([i] : D)$ **shift** $(a[i]@[i] \rightarrow a[i+c]@[i+c], overflow)$;

  The `shift` statement shifts elements of a from all VPs to their neighboring VPs by a constant offset *c*. $overflow$ specifies what to do at the boundary. It can either be *TRUNC* (truncated) or *WRAP* (wrap-around).

- Reflect

  with $([i] : D)$ **reflect** $(a[i]@[i] \rightarrow a[n-i-1]@[n-i-1])$;

  It flips the elements of a along the bisection of a dimension. The elements at the first and last VPs are swapped, the elements at the second and the second to the last VPs are swapped, and so forth. It can be thought as a collection of concurrent `swap` statements.

- Transpose

  with $([i,j] : D)$ **transpose** $(a[i][j]@[i,j] \rightarrow a[j][i]@[j,i])$;

  The `transpose` statement is defined only fro a two-dimensional square-shaped domain. It flips elements of a along one of the two diagonal lines of the domain.

- Skew

  with $([i,j] : D)$ **skew** $(a[i][j]@[i,j] \rightarrow a[i][i+j+c]@[i,i+j+c])$;

The **skew** statement is defined only for a two-dimensional domain. The above statement will skew the matrix a in row direction. The step is c. It corresponds to a collection of concurrent **shift** statements, each with a different offset.

Although most of the examples given for the abstract communication statements are based on one-dimensional domain, it is also fine to use them on a two-dimensional domain. For instance, a copy statement

with $([i,j] : D)$ copy $(a[c_1][c_2]@[c_1,c_2] \rightarrow b[c_3][c_4]@[c_3,c_4])$;

copies $a[c_1][c_2]$ to $b[c_3][c_4]$ over a two dimensional domain.

However there are more features we can specify in a two dimensional domain. Let's take a look at the following copy statement,

with $([i,j] : D)$ copy $(a[i][c_2]@[i,c_2] \rightarrow b[i][c_4]@[i,c_4])$;

Notice the row indices in the source and destination expressions are variables. This means a concurrent row operation. Within each row, element c_2 of a is copied to element c_4 of b. The result is that column c_2 of matrix a is copied to column c_4 of matrix b. Similarly, we can specify concurrent column operation.

The data expressions in a communication statement can also be aggregate array data. For instance,

with $([i] : D)$ copy $(a[c_1][:]@[c_1] \rightarrow b[c_2][:]@[c_2])$;

Only only one dimension (row) of matrices a and b is distributed over a one dimensional domain D. The above statement will copy row c_1 of matrix a to row c_2 of matrix b. The colon notation is borrowed from FORTRAN 90 to indicate a whole array dimension

II.4 PARALLEL CONSTRUCT

Parallel loops are often suggested as an easy method for programmers to express the parallelism in an algorithm in a structured way. In ACE, we use `forall` loop as a parallel construct. It looks something like this:

forall ($[i, j]$: D)
statement

`forall` loops can contain or be contained in other sequential or parallel loops.

The meaning of a parallel loop can be very different depending on how its semantics are interpreted. A `forall` loop in ACE is used when there are no dependence relations between iterations; thus the iterations of the loop can be executed in any order, including in parallel. The `forall` body can consist of multiple statements. These statements are treated as a whole block. They have to be executed in the order they are presented, since there can be dependencies inbetween the statements. But the loop body as a block of a certain iteration has no dependence relationship with any other iterations.

As an example, Figure 2(a) shows a matrix multiply algorithm. The two statements used to calculate `c[i][j]`, initialization and `k` loop, have to executed in that order. Otherwise the result will be incorrect. However, there is no dependence relation between the iterations of `i` and `j` loops. They can be executed in any order. We can use `forall` loop to express that. Assuming the data arrays are aligned to a two dimensional domain D, the parallelized algorithm is shown in Figure 2(b).

II.5 COLLECTIVE COMMUNICATION LIBRARY

The low-level communication interface in our model is a library of message-passing routines, called *the collective communication library* or CCL [12]. These communication routines are defined with respect to an abstract processor network

```
for (i=0;i<n;i++)
  for (j=0;j<n;j++) {
  c[i][j] = 0.0;
  for (k=0;k<n;k++)
    c[i][j] += a[i][j]*b[i][j];
  }
```

(a)

domain $D = [1 : n\ ,\ 1 : n]$;
forall $([i, j] : D)$ {
  $c[i][j] = 0.0$;
  for $(k = 0; k < n; k + +)$
    $c[i][j] += a[i][k] * b[k][j]$;
}

(b)

Figure 2. Parallelize Matrix Multiply Using `forall` Loop.

(a one-dimensional array or a two-dimensional mesh) and are pre-implemented on actual target machines.

The CCL routines cover the same spectrum of communication patterns and scopes of the abstract communication statements. However, the CCL routines represent communications at a lower level. One major difference between the two levels is that the input and output data to a CCL routine is represented by two pointers to two sequential data buffers, regardless of the dimensionality of the original data structures.

A typical CCL routine has three forms: one for one-dimensional processor network, one for two-dimensional, and one for concurrent row/column. They take the following general form:

ccl_xxx1d($pinfo, ..., x, xcnt, elt\_size, y, ...$)
ccl_xxx2d($pinfo, ..., x, rcnt, ccnt, elt\_size, y, ...$)
ccl_xxxrc($pinfo, dim, idx, ..., x, rcnt, ccnt, elt\_size, y, ...$)

where $pinfo$ is a structure containing system, domain, and data distribution related information, such as the location of the processor in the mesh (*my_node, my_row* and *my_col*); x and y are pointers to input and output data buffers, respectively; $xcnt$ is the number of elements and elt_size is the size of an element; $rcnt$ and $ccnt$

represent the number of elements is row and in column in a two dimensional array.

A list of CCL routine forms and their functionality can be found in [12].

CHAPTER III

COMPILATION APPROACH

The abstract communication statements need eventually be transformed into lower level communication statements. Since most of the information is already in the explicit abstract communication statements, the major challenge is to incorporate domain decomposition and to manipulate data in the message buffer.

As illustrated in Figure 3, an ACE program can be transformed in two directions. One is to transform it into a message-passing program that can be compiled and executed on a distributed-memory machine (i.e. iPSC/2). The other direction is to transform it into a sequential C program that will run on a conventional sequential machine.

An ACE program is parsed and a syntax tree is generated. The syntax tree is traversed once to generate a symbol table. After the symbol table is generated, the syntax tree is then passed to the transformation. The transformation will generate a new syntax tree based on the syntax tree it receives. Mainly three parts of the original program will be changed once the transformation is done. The first part is the declaration part. The second part is the communication statements. The third part is the parallel loops. The rest of the program will stay unchanged, and all the node structures from the original syntax tree are simply copied to the transformed syntax tree. The three parts of transformation are discussed in detail in the following sections.

This chapter will discuss the above approaches in more detail. Section III.1

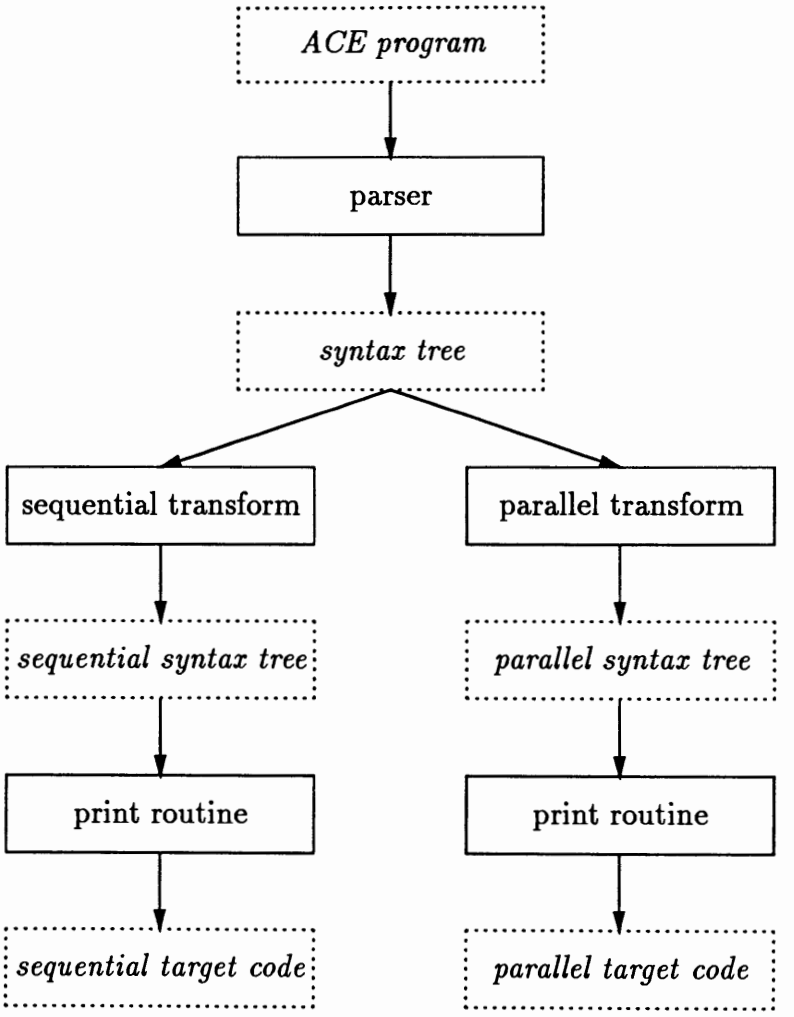


Figure 3. The Evolution of An ACE Program To Target Code.

discusses how a syntax tree and a symbol table is generated. The transformation of declaration statements, communication statements and parallel loops are discussed in sections III.2, III.3 and III.4. Section III.5 is about I/O issues. Section III.6 talks about code generation. Section III.7 discusses the issues of how to compile an ACE program to a sequential C program.

III.1 SYNTAX TREE AND SYMBOL TABLE

The ACE grammar is based on C grammar [8]. Changes are made to reflect the extensions added. These extensions can be divided into the following three categories, data distribution, data movement and parallel loops. The ACE parser is generated using UNIX tools *Lex* and *Yacc*. For every reduction rule in the grammar, there is a corresponding node structure. The action for each reduction makes a new node structure which contains pointers to subnodes. The pointer to this new node is then returned to its upper level node. A syntax tree is returned after an ACE program has been parsed.

Once the syntax tree has been generated by the parser, it is traversed once to generate the entries of the symbol table. The symbol table only holds the information needed to do the transformation. The information we are interested in are those about domain variables and all the data variables that are aligned to domains (They are the ones being distributed). Domain variable information (name, size, etc.) is given in domain declarations. The data variables that are aligned are those declared in alignment declarations.

The symbol table is a linked list of unions called *symbol*. A *symbol* can be a structure of domain variable or a structure of data variable.

A *symbol* for a domain variable consists of the following fields,

- Identifier

  Domain name.

- Range

  The range of a domain.

Part (a) of Figure 4 shows a domain declaration and part (b) is its corresponding

domain $D = [1 : n, 1 : n]$;	Identifier: D Range: $[1 : n, 1 : n]$
(a)	(b)

Figure 4. Symbol Table Entry of A Domain Variable.

symbol table entry.

A *symbol* for a data variable consists of the following fields,

- Identifier

  Variable name.

- Type

  The type of the data (e.g. int, float).

- Domain

  If the data is aligned, this is the domain that it's aligned to. If it is not aligned, its value is NULL.

- Alignment

  It is an integer array. Each element corresponds to a dimension of the data array, and its value reflects the alignment of the dimension. If the dimension is aligned to the first dimension of the domain, the value is 1. If the dimension is aligned to the second dimension of the domain, the value is 2. If the dimension is not aligned, the value is -1. (We are assuming that the dimensionality of a processor domain does not exceed two.)

- Size

  It is a list of expressions. Each of them is the corresponding dimension size of

	Identifier: a Type: **double** Domain: D
double $a[100][N][M]$; **align** $a[i][j][:]$ **to** $[i,j] : D$;	Alignment: 1 2 − 1 Size: 100 N M Dimensionality: 3 Aligned: *TRUE*
(a)	(b)

Figure 5. Symbol Table Entry of A Data Variable.

the data array.

- Dimensionality

  It is an integer that indicates the dimensionality of the data array.

- Aligned

  It is a boolean that indicates whether the data array is aligned (therefore possibly distributed). Only those data arrays that are distributed need extra attention in transformation stage.

Part (a) of Figure 5 shows a data variable declaration and its align declaration. Part (b) shows the corresponding symbol table entry for this variable.

The key to each entry of the symbol table is *identifier*. Since data variables can not be aligned to two domains, and domain variables cannot be redeclared, it is safe to assume that the identifiers in symbol table are unique.

In the mean time, some other important information is also stored. The shape and the size of the physical processor network is critical for data distribution. This information is obtained from the `distribute` statement and is stored in a global variable for later use.

III.2 DECLARATION STATEMENT TRANSFORMATION

Domain declaration, alignment declaration and distribution declaration in an ACE program are used to define the virtual domain, data array alignment and distribution scheme. They hold critical information for distributing data arrays among node processors. As discussed in the previous section, the information is already stored in the symbol table. These statements are simply omitted after transformation.

As far as variable declarations are concerned, if a data variable declared in the original program is not distributed, nothing needs to be changed, the original declaration stays the same. If, however, a data array is distributed, it means each processor node will only have a subset of the whole array. The size of the subarray is only a fraction of the total size. These variables are redeclared, and their spaces are reallocated.

Assuming the distribution is block distribution and the total number of nodes always divide the original array size (so that the data array can be evenly distributed among the nodes), the size of the portion that is local to the node is the original array size divided by the number of nodes. In case of a one dimensional processor network, the local size is represented by '`_cnt`'. If the processor network is two dimensional, the local size of the dimension distributed along the row is '`_rcnt`', and the local size of the dimension distributed along the column is '`_ccnt`'.

A distributed data array is redeclared. If it is a one dimensional array, then it is declared as a pointer. If it is a two dimensional array, then it is declared as a pointer to pointers. And so on. The dynamic allocation of these data arrays are done through some predefined functions. For example, function `alloc_double_2d_array(a,b)` will return a pointer to a two dimensional array of

type `double` with dimension sizes `a` and `b`. This allocation function has other types and size combinations like `alloc_float_1d_array` and so on.

As an example of showing how it looks, Figure 6(b) is a program section that the compiler will generate if the declaration of the original program is like part (a).

At the beginning of the program, a new variable _pn is introduced. (All variables that are compiler generated start with an underscore '_'). It is of type `struct ParInfo`. It has the following fields, providing the information of the underlying processor network.

- `num_nodes`

  The total number of processor nodes in the network.

- `my_node`

  Node id of the current node.

- `my_pid`

  The processor id of the program running on the node.

These values can be obtained from system calls on the target machine. In the case of iPSC/2, these calls are: `numnodes()`, `mynode()` and `mypid()`.

In the case of a two dimensional processor network (mesh), the following fields are also used,

- `row_nodes`

  The number of nodes in a row. This value is predefined somewhere in the program.

- `col_nodes`

  The number of nodes in a column. This value is also predefined somewhere in the program.

double $a[100][N][M]$;
align $a[i][j][:]$ to $[i,j] : D$;
distribute a over $[2,2]$;

(a)

```
double *a;
struct ParInfo _pn;
int _rcnt;
int _ccnt;

_pn.num_nodes = numnodes();
_pn.my_node = mynode();
_pn.my_pid = mypid();
_pn.col_nodes = 2;
_pn.row_nodes = 2;
_pn.my_col = _pn.my_node %
_pn.col_nodes;
_pn.my_row = _pn.my_node /
_pn.col_nodes;
_rcnt = 100/_pn.row_nodes;
_ccnt = N/_pn.col_nodes;

a =
alloc_double_3d_array(_rcnt,_ccnt,M);
```

(b)

Figure 6. Example of Declaration.

- `my_col`

  The column index that the node is in. If a simple block partitioning strategy is used, it is given by `my_col = my_node % col_nodes`.

- `my_row`

  The row index that the node is in. If a simple block partitioning strategy is used, it is given by `my_row = my_node/ col_nodes`.

III.3 COMMUNICATION STATEMENT TRANSFORMATION

The communication statements are a very important part of the ACE language. Each of the communication statements which occurs in the original program is transformed into a compound statement. This compound statement consists of a list of local variable initializations, data packing statements, a call to a Collective Communication Library (CCL) routine, and data unpacking statements.

III.3.1 Local Variable Initialization

A list of local variables are declared and initialized for a communication routine. These variables will be used later by the CCL library routine, data packing and unpacking. Generally speaking, there are five kinds of variables, variables to represent location information, variables to represent array subscripts, variables to represent data sizes, variables to represent data buffers and variables to represent loop indices.

Each of them is discussed in detail using the following communication routine as an example.

$$\mathsf{with}([i,j] : D)\ \mathsf{copy}(a[10,20]@[10,20] \rightarrow b[30,40]@[30,40])$$

- Location variables

  The origin and destination node id's are represented by variables `_loc1`, `_loc2` in a one dimensional processor network case. In a two dimensional case, every node is represented by its row id and column id. Thus the origin and the destination node id's are represented by `_iloc1`, `_jloc1` and `_iloc2`, `_jloc2`. The domain indices in the function call ([10, 20] and [30, 40]) are relative to domain D. They are not physical node id's. The actual node id is obtained by calling a predefined function `get_node_id`. For example, `get_node_id(a, &_pn)` will

return the physical node id domain index *a* resides on according to the information in structure _pn. In a two dimensional processor network, a node is represented by its row id and column id. Predefined functions `get_row_id()` and `get_col_id()` will return those values.

- Array subscripts

  The array subscripts in the above function call are global subscripts. They are assigned to variable `_c1` and `_c2` in a one dimensional case, and to `_c1`, `_c2`, `_c3` and `_c4` in a two dimensional case. The reason for this assignment is that these subscripts can also be complex expressions. The assignment will simplify later implementation.

  Since the data array is now distributed, the global subscripts can no longer be used to get to the intended array elements. They have to be transformed to local subscripts. Variables `_c1_local` and `_c2_local` are used as local subscripts if it's a one dimensional processor network. Variables `_c1_local`, `_c2_local`, `_c3_local` and `_c4_local` are used if it is two dimensional. The local subscript is the result of global subscript modula the local size of that data dimension.

- Data size variables

  The CCL library routine requires the number of bytes of the input data as a parameter. If the data is a scalar, the size is just the size of its type. If the communication involves replicated arrays, or if it is a row-wise or column-wise operation, the input and output data are both vectors, then the `_data_size` is `_dim_size*_type_size`.

- Data variables

  Two variables `_data1` and `_data2` are always used by CCL routines as the

```
int _iloc1, _jloc1, _iloc2, _jloc2, _c1, _c2, _c3, _c4,
    _c1_local, _c2_local, _c3_local, _c4_local,
    _data_size, _dim_size, _type_size, _k;
double *_data1, *_data2;
_c1 = 10;
_c2 = 20;
_c3 = 30;
_c4 = 40;
_iloc1 = get_row_nid(_c1, &_pn);
_jloc1 = get_col_nid(_c2, &_pn);
_iloc2 = get_row_nid(_c3, &_pn);
_jloc2 = get_col_nid(_c4, &_pn);
_c1_local = _c1 % _rcnt;
_c2_local = _c2 % _ccnt;
_c3_local = _c3 % _rcnt;
_c4_local = _c4 % _ccnt;
_dim_size = 1;
_type_size = sizeof(double);
_data_size = _dim_size * _type_size;
_data1 = malloc(_data_size);
_data2 = malloc(_data_size);
```

Figure 7. Local Variable Initialization.

input and output data buffer. They are declared as pointers to the data type. Their spaces are allocated with appropriate size.

- Loop indices

  Loop index _k is declared for use in packing and unpacking replicated data array. For simplicity reasons, it is always declared. Sometimes it is not used.

As an example, Figure 7 shows what local variables are declared and how they are initialized for the example listed at the beginning of this section.

III.3.2 Communication Patterns

Based on the number of nodes involved in sending and receiving the data, the communication routines can be categorized into the following four communication patterns,

- One-to-one

  Data messages move from one node of the processor array to another node. Only one node will send the message and one node will receive it. One-to-One communication routines include *copy* and *swap.*

  However, *swap* is a special case. Although there are only two nodes involved in the communication, both of them will send data and receive data.

- One-to-many

  Data messages move from one node to all other nodes in the processor array. One node will send the message but all nodes will receive it. These types of routines include *spread* and *scatter.*

- Many-to-one

  Data messages move from all the nodes in the processor array to one node. All the nodes will send a message, but only one will receive them. These types of routines include *reduce*, *search* and *gather.*

- Many-to-many

  Data messages move from all the nodes in the processor array to all nodes. All the nodes are involved in sending and receiving. These types of routines include *shift*, *reflect*, *transpose* and *skew.*

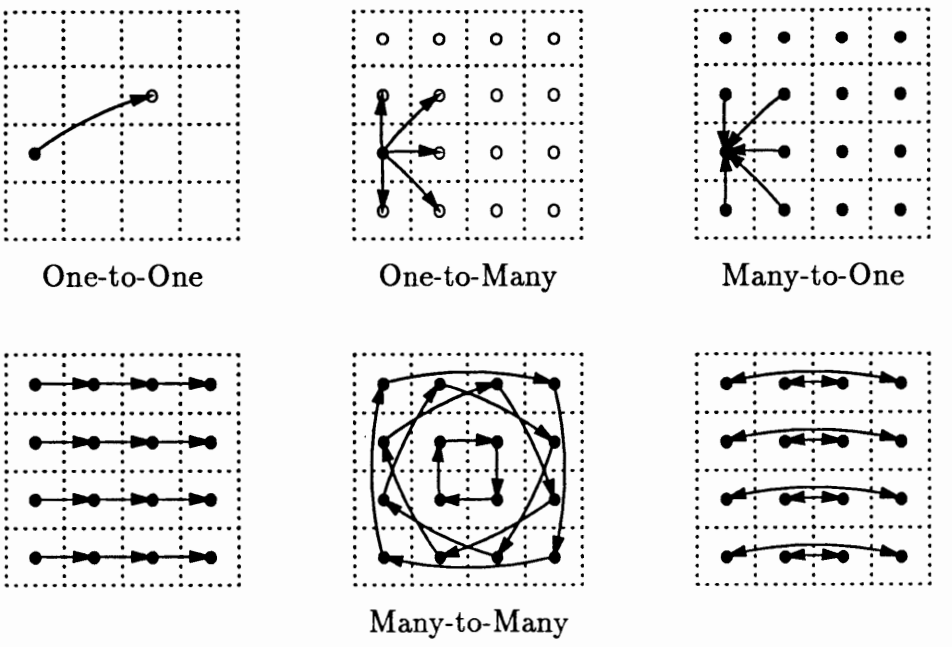


Figure 8. Four Basic Communication Patterns.

III.3.3 Code Structure

A communication routine is transformed into a compound statement. It does, mainly, three things, packing data, calling CCL library routine, and unpacking data. The origin node, which will send the data, packs all data elements it wants to send into a single message buffer `_data1`. The CCL library routine is then called. The resulting data is stored in buffer `_data2` on the destination node. `_data2` is unpacked, and all data elements are assigned to where they should go.

For different communication patterns, the structures of target code are different.

One-to-one communication routines involve two nodes. It may happen that the two virtual nodes are actually mapped onto one physical node, therefore no communication is needed. If they are two different nodes, a CCL routine has to

```
if (predicate) then
    if (node_test) then
        if (local_test) then
            local_assignment()
        else
            if (origin_test) then
                pack_data
            endif
            ccl_one_to_one_comm()
            if (destination_test) then
                unpack_data
            endif
        endif
    endif
endif
```

Figure 9. One-to-One Communication.

be called. The code structure for One-to-one communication routines is shown in Figure 9.

The meaning of the variables and procedures are explained as follows,

- *predicate*

  This is the predicate expression taken from the communication expression in the original function call. It is optional. If *predicate* is empty, the *if-then* structure is omitted.

- *node_test*

  This is to see if the node is involved in the communication. If yes, it will do the more work, otherwise it will skip this part. This test is only needed when the communication is one-to-one. In other communication patterns, all nodes

are involved.

A *node_test* for a one dimensional processor network may look like this:

```
((_pn.my_node == _loc1) || (_pn.my_node == _loc2))
```

- *local_test*

  This test is to see if the two locations are actually on one physical node. For the same reason as for *node_test*, this is only needed in one-to-one communication pattern. If the test is satisfied, the two locations are on the same node, all it has to do is some local assignments. No communication is necessary.

  A *local_test* for a two dimensional processor network may look like this:

  ```
  ((_iloc1 == _iloc2) && (_jloc1 == _jloc2))
  ```

- *local_assignment*

  The two data elements are on the same node. A simple assignment statement will do the job of copying, a couple more will do the job of swapping.

- *origin_test, destination_test*

  Boolean expression which is a test to see if the node is an origin node, or a destination node.

  An *origin_test* for a one dimensional processor network may look like this:

  `(loc1 == pn.my_node)` A *destination_test* for a two dimensional processor network may look like this:

  ```
  ((_iloc2 == _pn.my_row) && (_jloc2 == _pn.my_col))
  ```

- *ccl_comm*()

  This is the corresponding CCL library routine that was discussed in Chapter 2.

- *pack_data, unpack_data*

```
if (predicate) then
    if (origin_test) then
        pack_data
    endif
    ccl_one_to_many_comm()
    unpack_data
endif
```

Figure 10. One-to-Many Communication.

These are the code sections that do data packing and unpacking. The input data has to be packed into a single array buffer that is pointed by `_data1` before calling the communication routine. The output data is stored in buffer `_data2`. These are discussed in more detail in the following section.

Figure 10 is the code structures for one-to-many communication statements. Since all nodes will participate at the receiving end, there is no *destination_test.*

Figure 11 is the code structures for many-to-one communication statements. All nodes will participate at the sending end, there is no *origin_test.*

Figure 12 is the code structures for many-to-many communication statements. All nodes will participate at both sending and receiving ends, no *destination_test* or *origin_test* is needed.

III.3.4 Data Packing and Unpacking

Data packing and unpacking is another major issue. The collective communication routines only take a pointer to the input data and a pointer to the output data as parameters. If the data to be sent or received is not a scalar value, and involves more data elements, they have to be properly packed together before the

```
if (predicate) then
    pack_data
    ccl_many_to_one_comm()
    if (destination_test) then
       unpack_data
    endif
endif
```

Figure 11. Many-to-One Communication.

```
if (predicate) then
    pack_data
    ccl_many_to_many_comm()
    unpack_data
endif
```

Figure 12. Many-to-Many Communication.

communication call, and properly unpacked after the call.

Generally speaking, the data that are passed to the communication routines can be divided into three groups, those that only deal with scalar values, those that deal with replicated array elements and those that deal with row-wise or column-wise operations. They all have to be packed into a single data buffer to passed to a CCL routine, and unpacked after the call is completed.

The strategies used for different groups of data are described below.

Scalar. In the following statement, we are dealing with data that are of scalar

values.

$$\textsf{with}([i] : D)\ \textsf{copy}(a[10]@[10] \rightarrow b[20]@[20]);$$

Data packing (unpacking) is simply an assignment statement that assigns the the array element to _data1 (_data2),

```
*_data1 = a[10];
...
b[10] = *_data2;
```

Since _data1 and _data2 are always defined as pointers, dereferencing is needed.

Replicated Array. In the following statement, data array expressions contain [:], which means that dimension is replicated on every processor.

$$\textsf{with}([i] : D)\ \textsf{copy}(a[10][:]@[10] \rightarrow b[20][:]@[20]);$$

The *copy* statement is copying the 11th row of array a on virtual processor 10 to the 21st row of array b on virtual processor 20. The task of packing is to put all the elements in buffer _data1,

```
for(_k = 0; _k < n; _k++)
    _data1[k] = a[10][_k];
```

Unpacking is similar, it will assign buffer _data2 to the 21st row of array b.

Concurrent Row/Column Operation. In the following statement, data arrays a and b are distributed across a two dimensional processor domain.

$$\textsf{with}([i,j] : D)\ \textsf{copy}(a[10][j]@[10,j] \rightarrow b[20][j]@[20,j]);$$

In order to copy the 11th row of a to the 21st row of b, a concurrent column operation is needed. Every column of the virtual processor domain will copy its 11th element of array a to 21st element of array b.

Data packing looks like the following,

```
for(j = 0; j < _ccnt; j++)
    _data1[j] = a[10][j];
```

It will put all wanted elements that are local into buffer `_data1`. Unpacking is similar.

Combination of the Above. In the following statement, data array a is collected from all the nodes. It is then assigned to array c on processor 1.

$$\textbf{with}([i] : D)\ \textbf{gather}(a[i][:]@[i] \rightarrow c[:][:]@[1]);$$

This is a combination of replacated and distributed dimensions. In this case, we always process (pack or unpack) the replicated dimension(s) first, then the distributed dimension(s). Otherwise we may get a wrong answer.

Suppose the array a is of size 4×4 and is distributed as in Figure 13(a). The first two columns are on physical processor 0 and the other two are on physical processor 1. After packing, `_data1`'s on processor 0 and 1 are:

$$P0 : \texttt{\_data1} \rightarrow 0, 1, 4, 5, 8, 9, 12, 13$$
$$P1 : \texttt{\_data1} \rightarrow 2, 3, 6, 7, 10, 11, 14, 15$$

The CCL `gather` routine appends these two together and return the result as `_data2`.

$$\texttt{\_data2} \rightarrow 0, 1, 4, 5, 8, 9, 12, 13, 2, 3, 6, 7, 10, 11, 14, 15$$

Unpack this with row major we get the array in Figure 13(b).

That is obviously a wrong answer.

However, if we pack the replicated dimension first (in this case, column) we have:

$$P0 : \texttt{\_data1} \rightarrow 0, 4, 8, 12, 1, 5, 9, 13$$
$$P1 : \texttt{\_data1} \rightarrow 2, 6, 10, 14, 3, 7, 11, 15$$

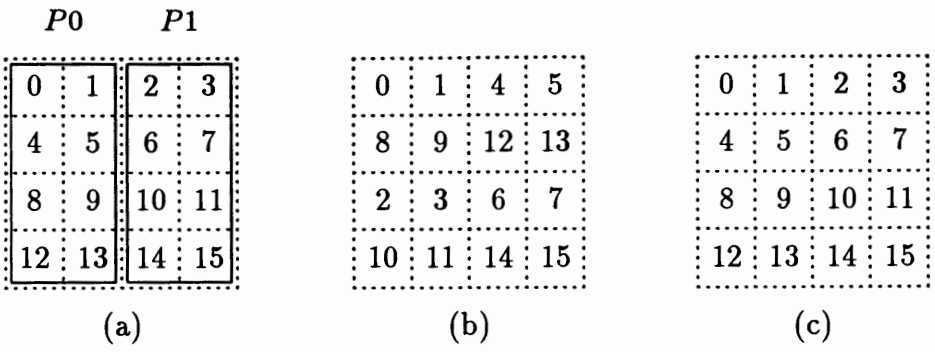


Figure 13. Problem With Packing And Unpacking Row Major.

After `gather` the resultant buffer is:

$$\_data2 \rightarrow 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15$$

Unpack the replicated dimension first, we get the matrix in Figure 13(c). That is the correct result.

`Swap` is a little different from others. Usually *data*1 is assigned a value at data packing time. It is then passed to the communication routine as an input data. The result from the communication is stored in *data*2. *data*2 is then assigned to its destination. Data packing (or unpacking) for `swap` involves both *data*1 and *data*2. *data*1 and *data*2 both have valid input data going into the communication routine and valid output data out of it.

III.3.5 Optional Domain Predicate

There may or may not be a domain predicate that appears in a communication statement. Three different cases may arise regarding domain predicate. Let's take a look at them, using the `spread` statement as an example: Three cases need to be considered,

- No domain predicate in a communication statement,

$$\textsf{with}([i,j] : D)\ \textsf{spread}(\alpha@[c_1,c_2] \rightarrow \beta@[i,j]);$$

The communication statement, in this case, **spread**, is always executed in the whole domain.

- Domain predicate exists in a communication statement. The variables referenced in the predicate are not domain index variables.

$$\textsf{with}([i,j] : D : \mathit{count} > 0)\ \textsf{spread}(\alpha@[c_1,c_2] \rightarrow \beta@[i,j]);$$

Here, the predicate, $\mathit{count} > 0$ dose not reference domain indices i or j. This **spread** operation is either executed in the whole domain (when $\mathit{count} > 0$) or not executed at all (when $\mathit{count} <= 0$).

- Domain predicate in the statement references domain indices.

$$\textsf{with}([i,j] : D : i > k)\ \textsf{spread}(\alpha@[c_1,c_2] \rightarrow \beta@[i,j]);$$

It means to spread α on node $[c_1, c_2]$ to a sub-domain of D where $i > k$ (all the rows that are greater than k).

To realize that, the **spread** operation is performed on every node as normal. But when it comes to data unpacking, only those β's on nodes that satisfy the predicate $i > k$ will have their values assigned, others will stay the same.

Things are a little different for **reduce** and **search**. The predicate has to be checked at data packing time. The approach taken is to replace the values on those VPs that don't qualify with a value that will not affect the outcome. For instance, if the operation is **reduce** and the operator is '+', then replace with a value of 0, if the operator is '*', then replace with a value of 1. If the

operation is to search for the maximum value, then the smallest number will be supplied to replace those that don't qualify, and so forth.

III.4 PARALLEL LOOP TRANSFORMATION

Parallel loops appear in ACE in the form of:

forall ([i , j] : D)
statement

It means for all i, j in the range of domain D, execute the *statement* in parallel. If there are n processors, the *statement* will be executed in parallel on all n processors. Since domain D is distributed across the processors, each processor is responsible for executing the *statement* on a subset of domain D. This is achieved by applying a regular `for` loop to *statement* on every processor.

The information of domain D is stored in the symbol table in an earlier stage. In order to generate `for` loops out of a `forall` loop, we need to extract range and distribute information of D from the symbol table. We assume that the way to distribute a domain is always block distribution. The current version of the compiler has only implemented block distribution scheme. Here we make another assumption that the ranges of a domain always start with 1. So the upper bound of a range represents the size of it. The subset that is to distributed on a processor is *upper_bound* / *pn.num_nodes*, in a one dimensional case. This is the `for` loop upper bound. In a two dimensional case, two nested `for` loops are generated. The outer loop is for row index (first dimension in the range) and the inner loop is for column index (second dimension in the range). The `for` loop upper bounds are *upper_bound*1 / *pn.row_nodes* and *upper_bound*2 / *pn.col_nodes* respectively.

The array indices that are referenced in *statement* may cause some problem. If an index variable is a left hand side value of an assignment, like in the following

array initialization statement.

$$\begin{array}{l} \textsf{forall } [i,j] : D \\ \quad a[i][j] = i * n + j; \end{array}$$

i and j on the left hand side are global values, not local values. These indices should be replaced by their global values. In the above case, i and j are replaced with `gi` and `gj`, with

$$\begin{array}{l} \texttt{gi} = \texttt{i} + (\texttt{n/num\_rows}) * \texttt{pn.my\_row}; \\ \texttt{gj} = \texttt{j} + (\texttt{n/num\_cols}) * \texttt{pn.my\_col}; \end{array}$$

If an index variable is simply referenced as an array subscript as in $a[i][j]$, (either on right hand side or left hand side) nothing needs to be changed. However if an index variable is assigned a new value inside the loop, or an array subscript is a linear or more complicated form of index variable (e.g. $a[3 * i][i + j]$). The transformation is not a trivial problem. It is not dealt with in this project.

The optional domain predicate in a `forall` loop is treated in a similar way as those in communication statements.

III.5 INPUT/OUTPUT

Input/output has always been a difficult part of a parallel language design, since I/O has not been well studied.

In this section, a simple approach to this problem is presented. The I/O functions discussed here will do more than reading and writing. After the data array is read in, it is distributed to all the nodes. Before the final result can be written out, it first collects subsections of the result from all the nodes. As far as the target machine, iPSC/2, is concerned, all the nodes in the cube are able to do I/O. All of them can read in data from a file or standard input, and write data to a file or standard output. We will always use node 0 in the cube to perform I/O. Before computation starts, node 0 will read in data arrays and distribute them to

all other nodes according to how they are aligned to the processor domain. After computation, all subsections of the result array that reside on different nodes are collected to node 0 and then printed out to the screen in its whole form.

The two functions we implemented are called `input` an `output`. They act just like function calls. The arguments supplied to the function are the names of data arrays that need to be read in or written out. The first parameter specifies the input stream. If it's '`stdin`', then the program will read in data from standard input. Otherwise, it is a file name, and the program will read in data from this file.

The ACE compiler will translate the `input` and `output` functions into a list of statements that will accomplish the goal. For every array name, a list of somewhat similar statements are generated. The information needed (e.g. the dimensionality and type of array, how it is distributed) is obtained from the symbol table.

A pre-defined function `input_`x`array` does the actual input and distribution. (x could be i, f or d for data types `integer`, `float` and `double`.) Inside this function, the whole array is read in on node 0. Node 0 then distributes the array to all other nodes in the cube. After the function call, the subsection of the original array is stored in a temporary buffer on each node. The local arrays get their values from these temporary buffers.

In the target code, `input_`x`array` may look like this:

```
input_iarray(pn, buf, rcnt, ccnt, dist)
```

The parameters are explained as follows:

- `pn`

  Type: `struct ParInfo` $\star$

  Holds information of the underlying processor network.

- `buf`

  Type: `int` $\star$

  For `input` routine: has values of a subarray after the call.

  For `output` routine: has values of a subarray before the call. The type of `buf` varies according to the actual function call. It is `float` $\star$ if the function is `input_farray`, or `double` $\star$ if the function is `input_darray`.

- `rcnt`

  Type: `int`

  Indicates number of rows in the subarray. *rcnt* has the value of 1 if the subarray is one dimensional.

- `ccnt`

  Type: `int`

  Indicates number of columns in the subarray.

- `dist`

  Type: `int`

  This is a flag indicating how the array is distributed. It can be one of the following four values:

  1 : A one dimensional array distributed across a one dimensional processor network.

  2 : A two dimensional array distributed across a two dimensional processor network.

  3 : A two dimensional array distributed row-wise across a one dimensional processor network.

```
#define n 16
dom spatial D = [1 : n, 1 : n];

main()
{
   double a[n][n];
   int b[n][n], i;
   align a[i][:], b[:][i] over [i]:D;

   input(a);
   ...
   output(b);
}
```

Figure 14. Usage of `input` And `output`.

4 : A two dimensional array distributed column-wise across a one dimensional processor network.

The same format is used for `output_xarray` routines. Things are pretty much just the opposite inside the routines. The subsections provided by `buf` on each node is collected to node 0 and printed out to the screen.

As an example, Figure 14 shows how `input` and `output` functions are used in the context of an ACE program. Figure 15 gives the program that the compiler will compile to.

III.6 TARGET CODE

The transformed syntax tree is printed out in C form by a print routine. The program can be compiled and run on the iPSC/2 machine.

Besides the CCL library and the I/O routines, another two sets of utility routines are also pre-written to make the compilation easier and the target code

```
#define n 16

main()
{
    struct ParInfo _pn;
    double **a;
    int **b;

    _pn.num_nodes = numnodes();
    _pn.my_node = mynode();
    _pn.my_pid = mypid();
    _cnt = n/_pn.num_nodes;
    a = alloc_2d_array(_cnt,n);
    b = alloc_2d_array(n,_cnt);

    {
        int _i,_j;
        double *_buf;

        _buf = malloc(_cnt*n*sizeof(double));
        input_darray(&_pn,_buf,_cnt,n,3);
        for (_i=0;_i<_cnt;_i++)
            for (_j=0;_j<n;_j++)
                a[_i][_j] = _buf[_i*n+_j];
    }

    ...

    {
        int _i,_j;
        int *_buf;

        buf = malloc(n*_cnt*sizeof(int));
        output_iarray(&_pn,_buf,n,_cnt,4);
    }
}
```

Figure 15. Target Code of input And output.

more readable.

- `get_xnid`

  It returns the node id of a node that define its location in a processor network. This set of routines include `get_nid`, `get_row_nid` and `get_col_nid`. Given the index of a virtual processor, they will return the actual physical node index that the virtual processor is assigned to.

- `alloc_`*type*`_n`*d*`_array`

  This is a set of routines that dynamically allocate storage space for different type and size of data arrays. Here, *type* is the data type, it can be `int`,`float` or `double`; *n* is the dimensionality of the array, it can be 1,2 or 3.

III.7 SEQUENTIAL TRANSFORMATION

An ACE program can also be compiled into a sequential C program that can be executed on a conventional sequential machine. The programmer doesn't have to write another program for sequential machines. It is also useful to help the programmer to check the correctness of an ACE algorithm without using a parallel machine.

The compilation to a sequential program is relatively trivial compared to the compilation to a parallel one. The focus, like the parallel version, is on the three extension parts, data distribution, data movement and parallel loops. This section discusses what needs to be done to these extensions in order to get a sequential program.

III.7.1 Data Distribution

Domain declaration, alignment declaration and distribution declaration statements in ACE are means for programmers to present information that are related to data distribution. On a sequential machine, we have a shared-memory environment. All data are stored in a global memory space. There is no need to distribute data. Therefore the information for data distribution can be ignored.

`Domain`, `align` and `distribute` statements are simply deleted from the syntax tree. However, there is one piece of information in these statements will be used later. That is the size of each dimension of a virtual processor domain. The lower and upper boundaries of the domain are needed to transform communication statements and parallel (`forall`) loops since they are defined on the domain. The information is stored in the symbol table.

III.7.2 Data Movement

In the sequential version, all data is stored in a global memory space. All processes are able to access them. The virtual processor domain that all communication statements are based upon becomes a single global domain. No communication will be needed even if a data reference is across the virtual processor domain.

Generally speaking, a communication statement can be replaced by an assignment statement. There are several variations for different communication pattern. They are discussed as following:

Communication Patterns. Different communication patterns will generate different loop bodies in sequential form.

- One-to-one

  The two statements that belong to this category are `copy` and `swap`. The transformation is pretty straight forward. A `copy` will result in a single as-

signment statement, while a **swap** statement will result in three assignment statements and a temporary variable.

- One-to-many

  This category has two statements, **spread** and **scatter**. Their sequential code segments consist of a **for** loop that iterates upon the domain.

- Many-to-one

  Reduce, **search** and tt gather statements belong to this category. Since they involve all VPs, a **for** loop is needed. For **reduce**, the destination expression should be appropriately initialized before reduction can take place. If the operator is '+', then the initial value is 0; if the operator is '*', then the initial value is 1; and so forth.

- Many-to-many

  In this category, the source and the destination expressions of a communication statement are usually different parts of a same data array. If this is actually the case, extra buffering is needed so that the old values and the new values of a data array are not confused. If the communication statement is **shift**, then its boundary condition has to be taken into acount.

Different Array Representations. The format of source and destination data, either it's a scalar, an aggregate array data or a row/column operation can make some difference, too.

- Scalar

  Nothing special needs to be done in this case.

- Aggregate array

  For every communication statement, if there is aggregate array involved, an-

with $([i,j] : D)$ copy($a[c_1][c_2] \rightarrow b[c_3][c_4]$); `b[c3][c4] = a[c1][c2]`

(a) Scalar

with $([i,j] : D)$ copy($a[c_1][c_2][:] \rightarrow b[c_3][c_4][:]$);

```
for (k=0; k<n; k++)
  b[c3][c4][k] =
a[c1][c2][k];
```

(b) Aggregate array

with $([i,j] : D)$ copy($a[i][c_2] \rightarrow b[i][c_4]$);

```
for (i=0; i<n; i++)
  b[i][c4] = a[i][c2];
```

(c) Concurrent row operation

Figure 16. Sequential Target Code of Different Data Representation.

other `for` loop is needed to take care of the elements in this dimension (the dimension represented by [:]). The ':'s are replaced by loop index '`k`'. The lower and upper bounds of the loop are those of that dimension.

- Concurrent row/column operation

  This case is similar to the previous one.

For each of the above, an example is shown in Figure 16 to demonstrate the basic ideas.

III.7.3 Parallel Loops

The `forall` loop used to express parallelism in parallel version does not have any meaning other than an iteration. Since the body statements of a `forall` loop are not supposed to have any data dependence relations inbetween the iterations, when executed sequentially, they can be executed in any order we like. To follow the C programming convention, we transform a `forall` loop into a `for` loop with its upper and lower bounds being those of virtual domain that the `forall` loop

operates on. The loop header is transformed from a `forall` loop to a `for` loop. Upper and lower bounds of the `for` loop is obtained from the domain information in the symbol table. If the domain is two dimensional, the first dimension will always be the outer loop. Again, all manipulation is done to the abstract syntax tree and the final result is printed out to a file in C form.

CHAPTER IV

EXPERIMENT

The current version of the compiler has implemented most features of ACE described in previous chapters, except the following: (1) it does not support cyclic distributions, (2) the size of the physical processor network always divides the size of the virtual processor domain, and (3) it doesn't provide extensive error checking or give out any error messages.

To use the ACE compiler, first store the ACE program in a file, preferably with a suffix `*.ac`. Then pass the file name to the compiler as a command line argument. There are two options that can be specified at the command line, `-p` and `-s`. The default is to generate code for a parallel machine. This specifies whether the user wants to generate target code for a parallel machine or a sequential machine. The target code file has suffix `*_par.c` or `*_seq.c` accordingly. For instance,

```
% ace -p test.ac
```

will generate target file '`test_par.c`' that can be executed on the iPSC/2.

```
% ace -s test.ac
```

will generate target file '`test_seq.c`' that can be executed on a sequential machine. At the same time, a file '`test.sym`' is generated which contains the symbol table information of the transformation.

This chapter shows a couple examples of well known parallel algorithms written in ACE. The examples illustrate how one can use ACE to express data distribution ,data movement and parallel constructs. In some cases, it can express a

parallel algorithm that is very hard for a sequential language to express. Readers will find ACE programs are usually shorter, cleaner and more understandable than their low-level message-passing counterparts.

The two examples given in this chapter are, a matrix multiply using Cannon's algorithm and a Gaussian elimination algorithm. The last section in this chapter shows some performance figures and analysis of the examples.

IV.1 CANNON'S ALGORITHM

Cannon's algorithm is an efficient parallel matrix multiplication algorithm. It multiplies two $n \times n$ matrices A, B and stores the result in a third matrix C on n^2 virtual processors. The algorithm can be described as two steps, initialization and computation.

1. Initialization

   The two input matrices are assumed to have been distributed over an $n \times n$ processor network. One matrix A is skewed row-wise and the other matrix B is skewed column-wise.

2. Computation

   The algorithm steps through n iterations. In each iteration, three steps are taken:

   (a) every processor performs a local multiplication and an addition.

   (b) array A is shifted to the left one step.

   (c) array B is shifted upwards one step.

The algorithm is illustrated in Figure 17. Let's look at c_{11}, During the first iteration, $c_{11} = c_{11} + a_{11} * b_{11} = a_{11} * b_{11}$. During the second iteration, A and

B are shifted. The elements a_{12} and b_{21} are on the same virtual processor as c_{11}. $c_{11} = c_{11} + a_{12} * b_{21} = a_{11} * b_{11} + a_{12} * b_{21}$. ... In the end c_{11} has the value of $a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} + a_{14} * b_{41}$.

Without using explicit communication statements, Cannon's algorithm is difficult to describe. The closest sequential algorithm can get is probably the following:

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        for (k=0;k<n;k++)
            c[i][j] += a[i][(i+j+k)%n]*b[(i+j+k)%n][j];
```

The abstract communication statements presented in ACE are perfect for describing Cannon's algorithm. Figure 18 shows the code in ACE for Cannon's algorithm. The matrices are distributed across a two dimensional processor network. After matrices A and B got their initial values, A is skewed row-wise by one step, and B is skewed column-wise by one step. They are accomplished by calling communication statement **skew**. In the computation stage, shifting of A and B is accomplished by **shift** statement.

The target code for this program is listed in Appendix A.

IV.2 GAUSSIAN ELIMINATION

Gaussian elimination is an algorithm used to reduce a matrix to one whose components of the diagonal and above remain nontrivial. (An upper triangular matrix.)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & b_1 \\ a_{21} & a_{22} & a_{23} & a_{24} & b_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & b_3 \\ a_{41} & a_{42} & a_{43} & a_{44} & b_4 \end{bmatrix} \Longrightarrow \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} & b'_1 \\ 0 & a'_{22} & a'_{23} & a'_{24} & b'_2 \\ 0 & 0 & a'_{33} & a'_{34} & b'_3 \\ 0 & 0 & 0 & a'_{44} & b'_4 \end{bmatrix}$$

(1) *Initialization*:

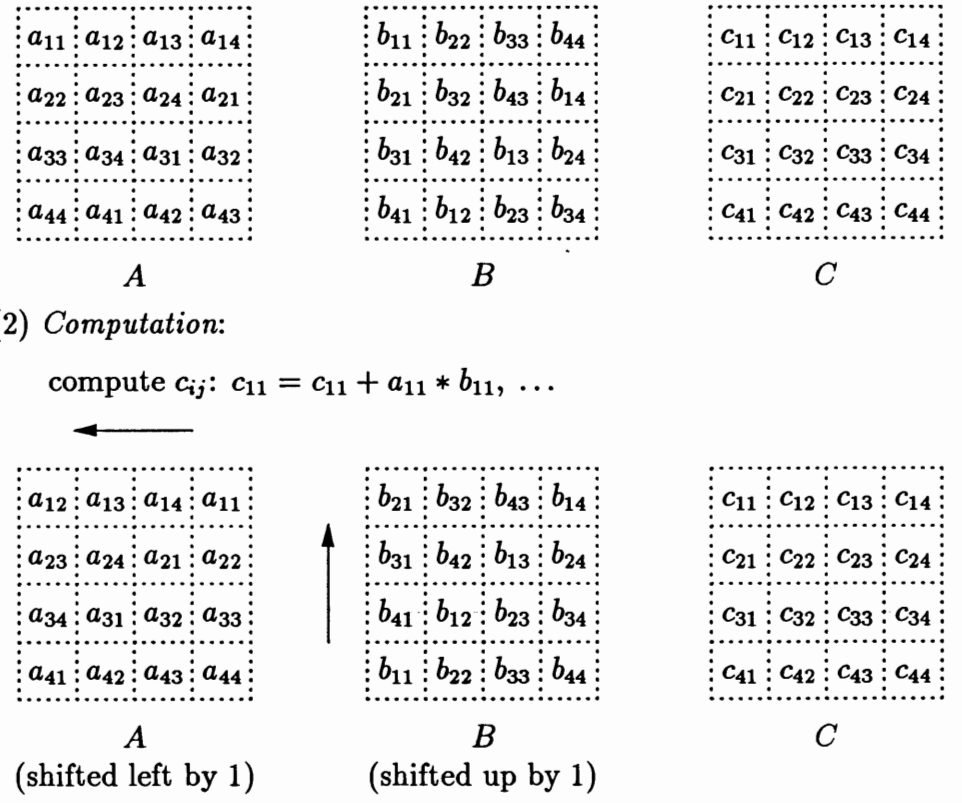


Figure 17. Illustration of Cannon's Algorithm.

Together with backsubstitution, it can be used to solve linear equations. If there are n equations with n unknowns x_i, $i = 0..n-1$. order all equations with all coefficients of the same unknown x_i in the same column i. For each successively smaller submatrix, scale all rows except top row by respective scale factor, so that left-most coefficient yields zero after subtracting. The resulting matrix is an upper triangle matrix. To compensate for limited precision on real machines, pivoting (rows and columns) or partial pivoting (rows) is often used.

In our example, the initial matrix is distributed row-wise across a one di-

```
#include < stdio.h >
#include "ccl.h"

#define n 8

domain D = [1 : n, 1 : n];

main()
{
   int a[n][n], b[n][n], c[n][n];
   int i, j, k;
   align a[i][j], b[i][j], c[i][j] to [i, j] : D;
   distribute D over [2, 2];

   input (a, b, c);

   with ([i, j] : D) skew(a[i][j] → a[i][i + j + 1]);
   with ([i, j] : D) skew(b[i][j] → b[i + j + 1][j]);

   for (k = 0; k < n; k + +) {
      forall ([i, j] : D)
          c[i][j]+ = a[i][j] * b[i][j];
      with ([i, j] : D)
         shift (a[i][j] → a[i][j − 1], WRAP);
      with ([i, j] : D)
         shift (b[i][j] → b[i − 1][j], WRAP);
    }

}
```

Figure 18. Cannon's Algorithm In ACE.

mensional domain. Every row resides on a different virtual processor. Gaussian elimination with partial pivoting can be expressed using ACE as in Figure 19. The algorithm works as follows: For each successively smaller sub-matrix (sequential loop k), do the following:

1. Find the largest element (absolute value) in the current column. The communication statement **search** will do exactly that. The pivot element is found and its index is stored in *pidx*.

2. Interchange the current row with the pivot row using **swap**.

3. Broadcast the current row to all other virtual processors. **spread** does that and the current row is stored in *pivot_row* on every VP for computation.

4. For every row below the current row, scale the pivot row by a factor *factor*, so that left-most element yields zero after subtracting.

The target code for this program is listed in Appendix B.

IV.3 PERFORMANCE ANALYSIS

A brief analysis of the performance of the above two programs is given in this section.

IV.3.1 Performance Measurement

The performance of each compiler-generated program is measured by two related parameters: *elapsed time* and *speedup*.

- Elapsed Time

  To measure the total elapsed time of a node program, timing statements are inserted into the program generated by the compiler. The initialization and output segments of the program are not timed.

- Speedup

  Speedup is defined as the ratio of the parallel execution time to the sequential

```
#include < stdio.h >
#include "ccl.h"

#define n 8

domain D = [1 : n];

main()
{
   double a[n][n + 1], pivot_row[n + 1], fac;
   int i, j, k, pidx;
   align a[i][:] to [i] : D;
   distribute D over [4];

   input(a);

   for (k = 0; k < n − 1; k + +) {
      with ([i] : D : i > k)
         search(a[i][k]@[i] → pivot_row[k]@[k], pidx, MAX);
      with ([i] : D)
         swap(a[k][:]@[k] → a[pidx][:]@[pidx]);
      with ([i] : D : i > k)
         spread(a[k][:]@[pidx] → pivot_row[:]@[i]);
      forall ([i] : D : i > k) {
         fac = a[i][k]/pivot_row[k];
         for (j = k; j < n + 1; j + +)
            a[i][j]− = pivot_row[j] ∗ fac;
      }
   }

}
```

Figure 19. Gaussian Elimination In ACE.

execution time. It shows how an application program scales with the size of a parallel machine.

Let T_1 denote the elapsed time of a program running on a single processor, and T_k be that on k processors. The speedup is computed as

$$S = \frac{T_1}{T_k}.$$

IV.3.2 Analysis

Both target programs have been compiled and executed on iPSC/2 successfully. Timing results for different matrix sizes are collected. The corresponding speedups are calculated.

Table I(a) shows the elapsed times of the Cannon's algorithm. Since a matrix multiply algorithm is of $O(n^3)$ complexity, every time n is doubled, the elapsed time will increase by eight times. This is clearly shown by each column of this table. The timing results are not optimal. Some more extensive testing has shown that in each case, a little more than half of the total elapsed time is spent on communication. Looking at the ACE program, we can see that `skew` is called twice to initialize the matrices. `shift` is called twice in each of the n iterations. In order to obtain good performance, the CCL library should be our point of focus for future research.

The timing results for Gaussian elimination in Table II (a) are more acceptable. This is because the communications involved, `search`,`spread` and `swap`, are more efficient than `shift`.

Part (b) of Tables I and II show the speedups of different matrix sizes. Their graphical representations are given in Figures 20 and 21. The numbers and the graphs show that the speedup is a lot better when the matrix size is large than when the matrix size is small. In the case of Cannon's algorithm, when the matrix size is 32, the speedup on 16 processors is 5.62; but when the matrix size is 128, the speedup jumps to 9.38, almost doubled. Similarly, in the case of Gaussian elimination, when

the matrix size is 32, we are actually experiencing a "speed-down"; but when the matrix size is 256, the speedup is 8.66.

With the increase of the number of the processors, each of them spends relatively more time communicating with each other. When the matrix size is small, each processor doesn't have much computation to do. The overhead of the communication is not compensated. When the matrix size is large, more time is spent on computation on each processor and the communication overhead becomes less of a factor.

TABLE I

PERFORMANCE OF CANNON'S ALGORITHM

Matrix Size	Elapsed Time (sec.)				
	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$
32x32	3.26	2.03	1.30	0.81	0.58
64x64	25.58	15.44	9.28	5.23	3.09
128x128	205.67	123.29	72.41	39.56	21.92

(a) Elapsed time on iPSC/2.

Matrix Size	Speedup				
	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$
32x32	1	1.61	2.51	4.02	5.62
64x64	1	1.66	2.76	4.89	8.28
128x128	1	1.67	2.84	5.20	9.38

(b) Speedup on iPSC/2.

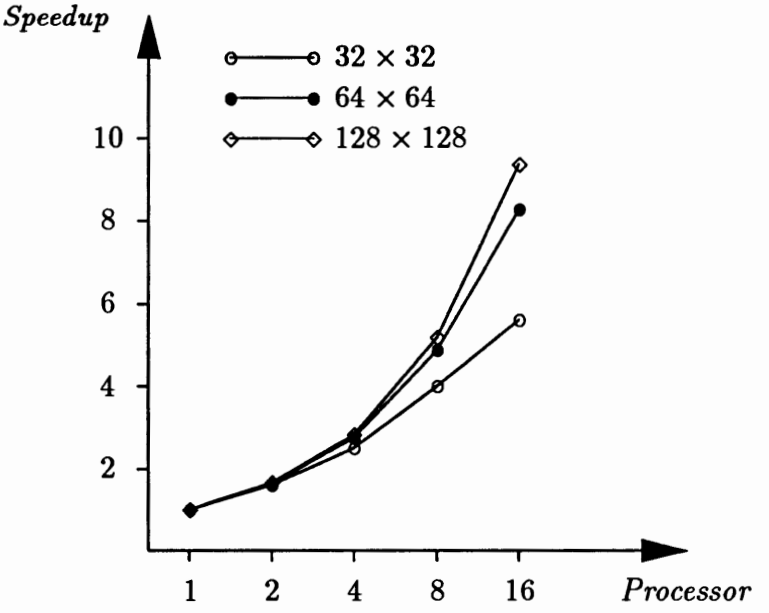


Figure 20. Speedup of Cannon's Algorithm On iPSC/2.

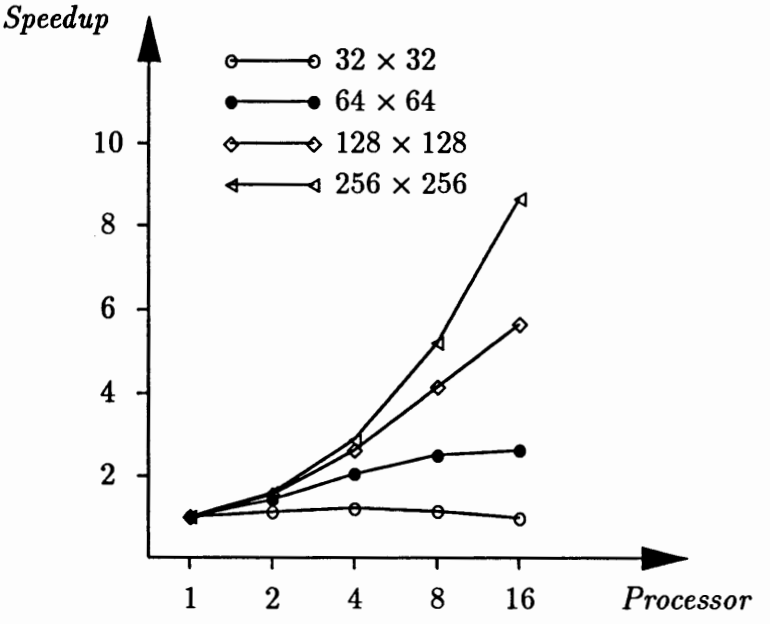


Figure 21. Speedup of Gaussian Elimination On iPSC/2.

TABLE II

PERFORMANCE OF GAUSSIAN ELIMINATION

Matrix Size	Elapsed Time (sec.)				
	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$
32x32	0.40	0.36	0.33	0.35	0.41
64x64	2.80	1.97	1.36	1.11	1.06
128x128	21.12	13.66	8.03	5.08	3.74
256x256	164.15	103.45	56.85	31.52	18.96

(a) Elapsed time on iPSC/2.

Matrix Size	Speedup				
	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$
32x32	1	1.13	1.21	1.14	0.98
64x64	1	1.42	2.06	2.52	2.64
128x128	1	1.55	2.63	4.16	5.65
256x256	1	1.59	2.89	5.21	8.66

(b) Speedup on iPSC/2.

CHAPTER V

CONCLUSION

The difficulty in programming distributed-memory machines is largely due to the need to explicitly manage data distribution and communication. Our research takes a modest approach to solving this problem. The message-passing language, ACE, introduced in this thesis gives the user a high-level representation of interprocessor communications. A set of abstract communication routines are defined based on data movement patterns. ACE is capable of describing actual parallel algorithms on message-passing machines, and it is portable across a class of machines.

V.1 RESEARCH SUMMARY

The main effort of this thesis is to show the advantages ACE language has over other its high-level or low-level counterparts. The thesis presented a design and implementation of an ACE compiler. Two well known parallel algorithms, namely Cannon's algorithm and Gaussian elimination, were programed in ACE, and they were compiled using the ACE compiler. The target programs were executed on an iPSC/2 multicomputer. Performance results were gathered and a brief analysis was given.

These examples show that ACE is very suitable for describing parallel algorithms. The ACE programs are easier to write and more understandable than programs using low-level message-passing primitives.

V.2 FUTURE WORK

The ACE compiler can be made more efficient and robust in many ways. The following are a few suggestions for future work.

- More Communication Routines

  At present, there are 12 abstract communication routines defined in ACE. (Refer to [11]) The corresponding low-level routines are also implemented on iPSC/2. In order to make ACE more powerful, more abstract communication routines are desirable. Matrix operation like linear transformation is one of them.

- More Distribution Schemes

  The current version of the compiler only supports block distribution. The data arrays can only be distributed across the processor network by blocks. Furthermore, the array dimension size has to be a multiple of the number of processors in that dimension. It will be feasible to have the compiler be able to handle cyclic distribution and uneven distribution and any combination of these.

- `forall` Loop Index Study

  As discussed in III.4, any reference of a loop index variable inside a `forall` loop has to be transformed. In this version, only very simple references are transformed. A more sophisticated transformation strategy has to be employed so that the `forall` loop can express more complicated computation.

REFERENCES

[1] M. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, January, 1986.

[2] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[3] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. Technical Report TR90-154, Dept. of Computer Science, Rice University, March 1991.

[4] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-indenpendent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors.* Elsevier,1991.

[5] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. Technical Report TR90-156, Dept. of Computer Science, Rice University, April 1991.

[6] P. Hatcher and M. Quinn, *Data-Parallel Programming.* The MIT Press, 1992.

[7] *iPSC/2 Users' Guide.* Intel Scientific Computers, Beaverton, OR, 1988

[8] B.W. Kernighan, D.M. Ritchie. *The C programming language* Prentice Hall, Englewood Cliffs, NJ, 1988.

[9] J. Li. Compiling Crystal for distributed-memory machines. PhD Dissertation, Dept. of Computer Science, Yale University, December 1991.

[10] J. Li. The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines. *Journal of Parallel and Distributed Computing*, 13:213-221, 1991.

[11] J. Li. Program distributed-memory machines with high-level communication abstractions. Technical Report, Dept. of Computer Science, Portland State University, September 1992.

[12] J. Li. *A manual for CCL Routines.* Technical Report, Dept. of Computer Science, Portland State University, October 1992.

[13] S. Otto. MetaMP: a higher level abstraction for message-passing programming. Dept. of Computer Science and Engineering, Oregon Graduate Institute, January 1991.

[14] M.J. Quinn, P.J. Hatcher, and J. Van Rosendale. Compiler C* programs for a hypercube multicomputer. In *Proceedings of the ACM/SIGPLAN symposium on Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, CT, July 1988.

[15] J. Rose and G. Steele Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[16] M. Wu and G. Fox. Compiling Fortran 90 programs for distributed-memory MIMD parallel computers. Technical Report CRPC-TR91126, Center for Research on Parallel Computation, Syracuse University, January 1991.

APPENDIX A

TARGET CODE FOR CANNON'S ALGORITHM

```
#include <stdio.h>
#include "ccl.h"
#define n 8

main(){
    float **a, **b, **c;
    int i, j, k;
    struct ParInfo _pn;
    int _rcnt, _ccnt;
    _pn.num_nodes=numnodes();
    _pn.my_node=mynode();
    _pn.my_pid=mypid();
    _pn.num_cols=2;
    _pn.num_rows=2;
    _pn.my_col=_pn.my_node%_pn.num_cols;
    _pn.my_row=_pn.my_node/_pn.num_cols;
    _rcnt=(n)/_pn.num_rows;
    _ccnt=(n)/_pn.num_cols;
    a=alloc_float_2d_array(_rcnt, _ccnt);
    b=alloc_float_2d_array(_rcnt, _ccnt);
    c=alloc_float_2d_array(_rcnt, _ccnt);
    {
        int _i,_j;
        float *_a;
        float *_b;
        float *_c;
        _a=malloc(sizeof(float)*_rcnt*_ccnt);
        input_farray(&_pn, _a, _rcnt, _ccnt, 2);
        for (_i=0;_i<_rcnt;++_i)
            for (_j=0;_j<_ccnt;++_j)
                a[_i][_j]=_a[_i*_ccnt+_j];
        _b=malloc(sizeof(float)*_rcnt*_ccnt);
        input_farray(&_pn, _b, _rcnt, _ccnt, 2);
        for (_i=0;_i<_rcnt;++_i)
            for (_j=0;_j<_ccnt;++_j)
                b[_i][_j]=_b[_i*_ccnt+_j];
        _c=malloc(sizeof(float)*_rcnt*_ccnt);
        input_farray(&_pn, _c, _rcnt, _ccnt, 2);
        for (_i=0;_i<_rcnt;++_i)
            for (_j=0;_j<_ccnt;++_j)
                c[_i][_j]=_c[_i*_ccnt+_j];
    }
    {
        int _offset, _data_size, _dim_size, _type_size, _k;
```

```
    float *_data1, *_data2;
    _offset=-1 ;
    _dim_size=1;
    _type_size=sizeof(float);
    _data_size=_rcnt*_ccnt*_type_size*_dim_size;
    _data1=malloc(_data_size);
    _data2=malloc(_data_size);
    for (i=0;i<_rcnt;++i)
    {
        for (j=0;j<_ccnt;++j)
            _data1[(i*_ccnt+j)]=a[i][j];
    }
    ccl_skewrc(&_pn, ROW, _data1, _rcnt, _ccnt, _type_size*_dim_size,
                _data2, _offset);
    for (i=0;i<_rcnt;++i)
    {
        for (j=0;j<_ccnt;++j)
            a[i][j]=_data2[(i*_ccnt+j)];
    }
}
{
    int _offset, _data_size, _dim_size, _type_size, _k;
    float *_data1, *_data2;
    _offset=-1 ;
    _dim_size=1;
    _type_size=sizeof(float);
    _data_size=_rcnt*_ccnt*_type_size*_dim_size;
    _data1=malloc(_data_size);
    _data2=malloc(_data_size);
    for (j=0;j<_ccnt;++j)
    {
        for (i=0;i<_rcnt;++i)
            _data1[(i*_ccnt+j)]=b[i][j];
    }
    ccl_skewrc(&_pn, COL, _data1, _rcnt, _ccnt, _type_size*_dim_size,
                _data2, _offset);
    for (j=0;j<_ccnt;++j)
    {
        for (i=0;i<_rcnt;++i)
            b[i][j]=_data2[(i*_ccnt+j)];
    }
}
for (k=0 ;k<n;k++)
{
```

```
    for (i=0;i<_rcnt;++i)
        for (j=0;j<_ccnt;++j)
            c[i][j]+=a[i][j]*b[i][j];
    {
        int _roff, _coff, _ovf, _data_size, _dim_size, _type_size, _k;
        float *_data1, *_data2;
        _coff=-1 ;
        _roff=0 ;
        _ovf=WRAP;
        _dim_size=1;
        _type_size=sizeof(float);
        _data_size=_rcnt*_ccnt*_type_size*_dim_size;
        _data1=malloc(_data_size);
        _data2=malloc(_data_size);
        for (i=0;i<_rcnt;++i)
            for (j=0;j<_ccnt;++j)
                _data1[(i*_ccnt+j)]=a[i][j];
        cl_shift2d(&_pn, _data1, _rcnt, _ccnt, _type_size*_dim_size,
                   _data2, _roff, _coff, _ovf);
        for (i=0;i<_rcnt;++i)
            for (j=0;j<_ccnt;++j)
                a[i][j]=_data2[(i*_ccnt+j)];
    }
    {
        int _roff, _coff, _ovf, _data_size, _dim_size, _type_size, _k;
        float *_data1, *_data2;
        _coff=0 ;
        _roff=-1 ;
        _ovf=WRAP;
        _dim_size=1;
        _type_size=sizeof(float);
        _data_size=_rcnt*_ccnt*_type_size*_dim_size;
        _data1=malloc(_data_size);
        _data2=malloc(_data_size);
        for (i=0;i<_rcnt;++i)
            for (j=0;j<_ccnt;++j)
                _data1[(i*_ccnt+j)]=b[i][j];
        ccl_shift2d(&_pn, _data1, _rcnt, _ccnt, _type_size*_dim_size,
                    _data2, _roff, _coff, _ovf);
        for (i=0;i<_rcnt;++i)
            for (j=0;j<_ccnt;++j)
                b[i][j]=_data2[(i*_ccnt+j)];
    }
}
```

```
}
```

APPENDIX B

TARGET CODE FOR GAUSSIAN ELIMINATION

```
#include <stdio.h>
#include <sys/types.h>
#include "ccl.h"
#define n 4

main(){
    double **a, pivot_row[n+1 ], fac;
    int i, j, k, pidx;
    struct ParInfo _pn;
    int _cnt;

    _pn.num_nodes=numnodes();
    _pn.my_node=mynode();
    _pn.my_pid=mypid();
    _cnt=n/_pn.num_nodes;
    a=alloc_double_2d_array(_cnt, n+1 );

    for (k=0 ;k<n-1 ;k++)
    {
        {
            int _loc2, _c2, _c2_local, _data_size, _dim_size, _type_size, *_index, _k;
            double *_data1, *_data2;
            _c2=k;
            _loc2=get_node_id(_c2,_cnt, &_pn);
            _c2_local=_c2%_cnt;
            _dim_size=1;
            _type_size=sizeof(double);
            _data_size=_dim_size*_type_size*_cnt;
            _data1=malloc(_data_size);
            _data2=malloc(_data_size);
            _index=malloc(_data_size);
            for (i=0;i<_cnt;++i)
                if ((i+_cnt*_pn.my_node)>k)
                    _data1[i]=a[i][k];
                else
                    _data1[i]=-999999;
                ccl_search1d(&_pn, _loc2, _data1, _cnt, _type_size*_dim_size,
                            _data2, _index, DOUBLE, MAX);
                {
                    pivot_row[k]=*_data2;
                    pidx=*_index;
                }
        }
        {
```

```
    int _loc1, _loc2, _c1, _c2, _c1_local, _c2_local,
        _data_size, _dim_size, _type_size, _k;
    double *_data1, *_data2, _temp;
    _c1=k;
    _c2=pidx;
    _loc1=get_node_id(_c1,_cnt, &_pn);
    _loc2=get_node_id(_c2,_cnt, &_pn);
    _c1_local=_c1%_cnt;
    _c2_local=_c2%_cnt;
    if ((_pn.my_node==_loc1)||(_pn.my_node==_loc2))
    {
        if (_loc1==_loc2)
            for (_k=0;_k<n+1 ;++_k)
            {
                _temp=a[_c1_local][_k];
                a[_c1_local][_k]=a[_c2_local][_k];
                a[_c2_local][_k]=_temp;
            }
            else
            {
                _dim_size=n+1 ;
                _type_size=sizeof(double);
                _data_size=_dim_size*_type_size;
                _data1=malloc(_data_size);
                _data2=malloc(_data_size);
                if (_pn.my_node==_loc1)
                    for (_k=0;_k<n+1 ;++_k)
                        _data1[_k]=a[_c1_local][_k];
                if (_pn.my_node==_loc2)
                    for (_k=0;_k<n+1 ;++_k)
                        _data2[_k]=a[_c2_local][_k];
                ccl_swap1d(&_pn, _loc1, _loc2, _data1, _data_size, _data2);
                if (_pn.my_node==_loc1)
                    for (_k=0;_k<n+1 ;++_k)
                        a[_c1_local][_k]=_data1[_k];
                if (_pn.my_node==_loc2)
                    for (_k=0;_k<n+1 ;++_k)
                        a[_c2_local][_k]=_data2[_k];
            }
    }
}
{
    int _loc1, _c1, _c1_local, _data_size, _dim_size, _type_size, _k;
    double *_data1, *_data2;
```

```
            _c1=k;
            _loc1=get_node_id(_c1,_cnt, &_pn);
            _c1_local=_c1%_cnt;
            _dim_size=n+1 ;
            _type_size=sizeof(double);
            _data_size=_dim_size*_type_size;
            _data1=malloc(_data_size);
            _data2=malloc(_data_size);
            if (_pn.my_node==_loc1)
                for (_k=0;_k<n+1 ;++_k)
                    _data1[_k]=a[_c1_local][_k];
            ccl_spread1d(&_pn, _loc1, _data1, _data_size, _data2);
            for (_k=0;_k<n+1 ;++_k)
                for (i=0;i<_cnt;++i)
                    if ((i+_cnt*_pn.my_node)>k)
                        pivot_row[_k]=_data2[_k];
        }
        for (i=0;i<_cnt;++i)
            if ((i+_cnt*_pn.my_node)>k)
            {
                fac=a[i][k]/pivot_row[k];
                for (j=k;j<n+1 ;j++)
                    a[i][j]-=pivot_row[j]*fac;
            }
        }
}
```